# DESCRIÇÃO

Divisão do código fonte na linguagem C em módulos e suas implementações pelas funções e procedimentos.

# PROPÓSITO

Compreender modularização, funções e procedimentos na criação de programas na linguagem C, passagem por valor e por referência e escopo das variáveis locais e globais.

# PREPARAÇÃO

Antes de iniciar o conteúdo deste tema, será necessário ter um compilador C instalado em sua máquina, como o GCC. Caso queira, instale o Dev-C++, que é um ambiente de

desenvolvimento integrado livre que utiliza os compiladores do projeto GNU para compilar programas para o sistema operacional Microsoft Windows.

Também será necessário possuir o arquivo com os códigos que serão utilizados neste tema. Clique aqui para fazer o download.

# OBJETIVOS

## MÓDULO 1

Descrever conceitos gerais de procedimentos e funções

## MÓDULO 2

Identificar os tipos de funções predefinidas na linguagem C

## MÓDULO 3

Distinguir passagem de parâmetros por valor e por referência

## MÓDULO 4

Localizar o escopo das variáveis locais e globais

# INTRODUÇÃO

Neste tema, demonstraremos como modularizar um programa na linguagem C usando funções e procedimentos, identificando os tipos de funções já predefinidas na biblioteca padrão do C. Relacionaremos, ainda, os tipos de passagem de parâmetros − por valor e por referência − identificando o escopo das variáveis como local e global.

# MÓDULO 1

---

- **Descrever conceitos gerais de procedimentos e funções**

# HISTÓRIA

## SAIBA MAIS

**Modularização** é o ato ou efeito de modularizar ou tornar modular, isto é, é o desenvolvimento, a construção ou a fabricação de algo em unidades ou peças autônomas que são passíveis de serem combinadas com outras para formarem um todo. No campo computacional, a modularização é empregada para dividir o programa em partes funcionais, partes essas que conversam umas com as outras.

Para entender melhor a definição de modularização, é necessário traçar um paralelo com a Revolução Industrial, que aconteceu na Inglaterra, no século XIX.

Antes da Revolução Industrial, todo produto era manufaturado, ou seja, construído inteiro, por um artesão e seu aprendiz. O sapateiro construía todo o sapato, desde a sola até o acabamento em couro, por exemplo. Para construir um conjunto de cadeiras, o marceneiro recebia a madeira e “esculpia” uma a uma.

Com o advento da Revolução Industrial, o trabalho passou a ser segregado. Uma pessoa construía uma parte da cadeira, − o assento, por exemplo − outra o encosto e outra montava. Tempos depois, a Ford chamou esse processo de linha de montagem, termo utilizado até hoje.

Dessa forma, um produto que era inteiriço passou a ser divido em módulos, que se interligavam e construíam o produto.

Na computação não foi muito diferente. O primeiro computador digital registrado, o ENIAC, não tinha o conceito de software, era tudo hardware. Porém, com a evolução da computação, em meados dos anos 1950, o cientista Von Neumann criou a arquitetura de computadores, cuja proposta inicial dividia o computador em 3 (três) partes:

## ENTRADA

## PROCESSAMENTO

## SAÍDA

Começava, assim, a Revolução da Computação. As máquinas de Von Neumann, como são conhecidas as que adotam esse modelo, dividem o hardware do computador em 3 partes:

**Unidade central de processamento.**

**Dispositivos de entrada/saída.**

**A memória.**

Dessa maneira, John Von Neumann introduziu o projeto lógico de computadores com programa armazenado na memória, modularizando o computador em partes como hardware e software.

Após essa fase, começaram a surgir os programas e o interesse comercial por computadores. Inicialmente, os programas eram escritos todos como um produto inteiriço, à semelhança de como era antes da Revolução Industrial. Sendo assim, algumas partes do código eram utilizadas mais de uma vez, o que deixava os programas com muita redundância no código, manutenção complicada e difícil, além da prática de utilizar os desvios com os " **Go To**".

No final da década de 1960, então, ocorreu a crise do software, devido às dificuldades enfrentadas no seu desenvolvimento frente ao rápido crescimento da demanda, à complexidade dos problemas a serem resolvidos e à ausência de técnicas bem estabelecidas para o desenvolvimento de sistemas.

Foi nessa época, mais precisamente em 1968, que ocorreu a Conferência da OTAN sobre Engenharia de Software (NATO – Software Engineering Conference) em Garmisch, Alemanha. O principal objetivo dessa reunião era estabelecer práticas mais maduras para o processo de desenvolvimento e, por essa razão, o encontro é considerado como o nascimento da disciplina de Engenharia de Software.

A criação da Engenharia de Software surgiu numa tentativa de contornar a crise e dar um tratamento de engenharia (mais sistemático e controlado) ao desenvolvimento de sistemas de software complexos.

Ao lado da Engenharia de Software surgiu a Programação Estruturada, que possibilitou dividir o código em pedaços especializados e acoplá-los mais tarde, produzindo, assim, o software que atendia às necessidades comerciais com mais facilidade.

Vale destacar que o processo é semelhante à Revolução Industrial do século XIX, só que no final da década de 1960 e década de 1970.

# CONCEITOS

Como já visto anteriormente, a modularização consiste em decompor um programa em uma série de subprogramas individuais. Trata-se de um método utilizado para facilitar a construção de grandes programas, através de sua divisão em pequenas etapas.

## COMENTÁRIO

A modularização segue o lema “dividir para conquistar”, ou seja, divide o problema em subproblemas menores, sucessivamente, conforme a necessidade.

A primeira etapa, por onde começa a execução do programa, é chamada de programa principal, e as demais são os subprogramas propriamente ditos, que são executados sempre que ocorre uma chamada, o que é feito através da especificação de seus nomes:

Programa principal:

É aquele por onde começa a execução, e chama, eventualmente, os demais subprogramas.

Subprograma:

É um programa que, geralmente, resolve um pequeno problema, e que está subordinado a um outro que solicitará seu acionamento. É possível que um subprograma chame outro subprograma.

# MAS POR QUE DEVEMOS MODULARIZAR?

A divisão do programa em módulos possui várias vantagens, entre elas:

Evitar que os programas fiquem grandes demais e difíceis de serem lidos e compreendidos.

Facilitar a leitura do código-fonte do programa.

Separar o programa em partes (blocos) que possam ser compreendidas de forma isolada (criação de módulos).

Evitar que um trecho seja repetido várias vezes dentro de um mesmo programa.

Permitir a alteração de um trecho do programa de forma mais rápida.

Utilizar um código em diferentes partes do programa, sem que ele precise ser escrito em cada local em que se deseje utilizá-lo.

Permitir o reaproveitamento de código já construído em outros programas (bibliotecas).

Ao dividir o programa em peças menores, será possível conectá-las para formar uma solução sempre que precisarmos resolver um problema mais complexo. Dessa forma, é necessário que essas peças sejam reutilizáveis, assim como as de lego.

Essas peças reutilizáveis podem ser chamadas de procedimentos e funções, que são formados por um bloco de código que executa uma determinada ação. De modo geral, cada função e cada procedimento deve realizar apenas uma ação, pois assim se tornam mais reutilizáveis e úteis para os programas.

# PROGRAMAÇÃO ESTRUTURADA

Inicialmente, os programas eram blocos lógicos, com início e fim, e compostos de estruturas de controle, como a de condição, seleção e repetição. Esses blocos consistiam em um conjunto de declarações e comandos delimitados pelas palavras início e fim, visando a aumentar a funcionalidade do programa.

Ao passo, porém, que os programas iam se tornando mais complexos, a divisão em partes menores os tornava mais claros e de fácil entendimento. Com o advento da programação estruturada, os programadores começaram a criar estruturas simples, usando as sub-rotinas.

Uma sub-rotina é uma ferramenta de programação que serve basicamente a dois objetivos:

Evitar que certa sequência de comandos que ocorra em vários locais de um algoritmo tenha de ser escrita repetidamente nesses locais.

Dividir a estrutura de um algoritmo em partes fechadas e logicamente coerentes.

As sub-rotinas devem codificar a solução para um problema pequeno e específico. Elas podem ser funções ou procedimentos (quando não retornam valores):

Os procedimentos são estruturas que agrupam um conjunto de comandos, que são executados quando o procedimento é chamado.

As funções são um tipo especial de procedimento, no qual, depois de executada a chamada, o valor calculado é retornado no nome da função, que passa a ser uma variável da expressão. Ou seja, elas sempre retornam algum valor para quem chamou, diferentemente dos procedimentos, que não retornam valor algum.

Portanto, em um arquivo de programa, funções/procedimentos conversam com o programa principal, e vice-versa.

# FUNÇÕES

As funções são procedimentos que retornam um único valor ao final de sua execução.

No exemplo abaixo, apresentamos alguns casos, como as funções **sqrt()**, **scanf()** e **printf()**, que calculam a raiz quadrada, leem um número da entrada padrão e imprimem um valor na saída padrão.

```
x = sqrt(4);
if (scanf ("%d", &x) == EOF)
printf("Fim de arquivo.\n");
```

## DECLARANDO UMA FUNÇÃO

Uma função em pseudolinguagem possui o seguinte formato:

```
funcao <nome-de-função>
[( <sequência-declarações-de-parâmetros>)]: <tipo-de-dado>
var
```

// Seção de Declarações Internas

inicio

// Seção de Comandos

fimfuncao

Toda função deve ter um tipo, que determina qual será o tipo de seu valor de retorno. Já os parâmetros de uma função determinam qual será o seu comportamento (como se fosse uma função matemática, na qual o parâmetro determina o valor da função). É possível, porém, que uma função não tenha parâmetros, basta não os informar. Na declaração a seguir, estão listados os parâmetros que serão utilizados na função:

tipo nome_da_funcao ( tipo < parametro1 >,
tipo < parametro2 >, . . . , tipo < parametron >) {
comandos;
return( valor de retorno );
}

É aconselhado declarar as funções antes do programa principal, mas o que ocorreria se declarássemos depois?

Embora aparentemente funcione, isso não pode ser feito na prática, pois alguns compiladores simplesmente não aceitam essa sintaxe, obrigando que a declaração ocorra antes da **função main** (ou melhor, antes de qualquer outra função que utilize uma determinada função).

**A função main** é o chamado programa principal. Ela apenas é uma função especial, que é invocada automaticamente pelo programa quando esse inicia sua execução

A saída para a questão citada anteriormente seria fazer uma declaração da função antes da função **main()**, mas colocar a sua implementação depois dela. Declarar uma função sem a sua implementação é muito semelhante a declará-la com a implementação. Substituímos as chaves e seu conteúdo por ponto e vírgula, ou seja:

tipo nome_da_funcao ( tipo < parametro1 >,
tipo < parametro2 >, . . . , tipo < parametron >);

No exemplo a seguir, em linguagem C, somente declaramos a função **imprime()**, sem implementá-la:

void imprime (int numero);

Com isso, separamos a declaração da implementação, permitindo que ela possa vir em qualquer lugar do código (antes ou depois de qualquer outra função). Além disso, um programa que declara todas as funções antes de usá-las tende a ser mais claro, pois o programador já sabe qual o conjunto de funções que ele pode usar, sem se preocupar com a forma como elas foram implementadas (ou sequer como elas foram desenvolvidas, caso o código esteja sendo desenvolvido por uma equipe).

# EXEMPLO DE FUNÇÃO

No exemplo a seguir, em linguagem C, identificamos a função soma() que realiza a soma de dois valores, que são passados por parâmetro:

```
int soma (int a, int b) {
return (a + b);
}
```

A expressão contida dentro do comando **return** é chamada de valor de retorno, e corresponde à resposta de uma determinada função. Esse comando é sempre o último a ser executado por uma função, e nada após ele será executado.

Quando utilizado, o comando return informa ao sistema operacional se o programa funcionou corretamente ou não. O padrão é que um programa retorne zero caso tenha funcionado corretamente, ou qualquer outro valor caso contrário.

# INVOCANDO UMA FUNÇÃO

Uma forma clássica de realizarmos a invocação (ou chamada) de uma função é atribuindo o seu valor a uma variável.

```
variavel = funcao (parametros);
```

Na verdade, podemos invocar uma função em qualquer lugar onde faríamos a leitura de uma variável, mas nunca a escrita. Vejamos o exemplo a seguir

```
printf ("Soma de x e y: %d\n", soma(x, y));
```

Nesse exemplo, ao executar a função **printf** será invocada a função soma, que irá calcular a soma das variáveis x e Y. Ao retornar a função, o resultado será apresentado na tela.

As variáveis passadas como parâmetros indicam os valores com os quais a função irá trabalhar. Esses valores são copiados para os parâmetros da função, que pode manipulá-los.

Os parâmetros passados pela função não necessariamente possuem os mesmos nomes que os que a função espera. Esses parâmetros serão mantidos intocados durante a execução da função.

O tipo void

O void é um tipo especial, utilizado principalmente em funções. Ele representa o “nada”, ou seja, uma variável desse tipo armazena conteúdo indeterminado, e uma função desse tipo retorna um conteúdo indeterminado.

# PROCEDIMENTOS

São estruturas que agrupam um conjunto de comandos, que são executados quando o procedimento é chamado, ou seja, procedimentos em linguagem C são funções do tipo void.

## DECLARANDO UM PROCEDIMENTO

Um procedimento possui o seguinte formato:

```
procedimento <nome-de-procedimento>
[(<sequência-de-declarações-de-parâmetros>)]
var
// Seção de Declarações Internas
inicio
// Seção de Comandos
fimprocedimento
```

Por exemplo, o procedimento a seguir imprime o número que for passado para ele como parâmetro:

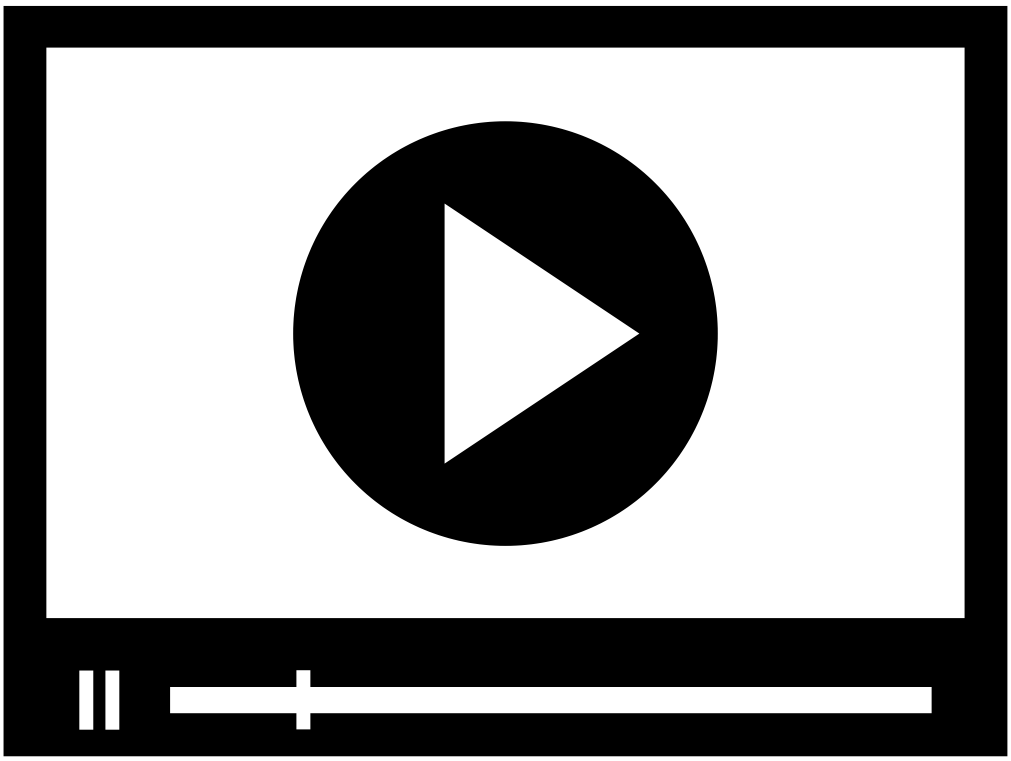
```
void imprime (int numero) {
printf ("Número %d\n", numero);
}
```

Ao se ignorar o valor de retorno de uma função e, para esta chamada, ela será equivalente a um procedimento.

## INVOCANDO UM PROCEDIMENTO

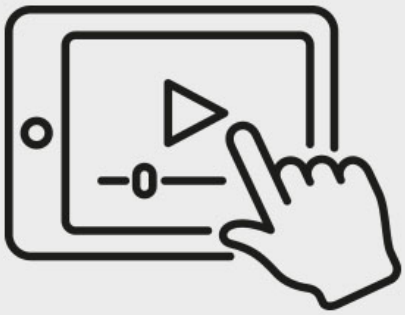
Para invocarmos um procedimento, devemos utilizá-lo como utilizaríamos qualquer outro comando, ou seja:

procedimento (parametros);



# FUNÇÕES E PROCEDIMENTOS NA LINGUAGEM C

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



# VERIFICANDO O APRENDIZADO

## 1. ESTUDAMOS SOBRE MODULARIZAÇÃO NA PROGRAMAÇÃO, QUE É A DIVISÃO DO CÓDIGO DO PROGRAMA EM SUB-ROTINAS. ASSINALE A ALTERNATIVA QUE CONTENHA TIPOS DE SUB-ROTINAS:

**A)** Pilhas e filas

**B)** Sub-rotina local e global

**C)** Procedimento e função

**D)** Passagem por valor e por referência

**E)** Matrizes e vetores

## 2. ASSINALE A ALTERNATIVA QUE CONTENHA A AFIRMATIVA CORRETA SOBRE FUNÇÃO E PROCEDIMENTO:

**A)** Não existe diferença entre função e procedimento.

**B)** As funções são um tipo de procedimento e podem ser usadas apenas uma vez no código do programa.

**C)** As funções sempre retornam algum valor a quem as chamou, e os procedimentos não retornam valor algum.

**D)** Os procedimentos sempre retornam algum valor a quem os chamou, e as funções não retornam valor algum.

**E)** Os procedimentos são um tipo de função com o objetivo apenas de chamar uma ou várias funções.

---

## GABARITO

**1. Estudamos sobre modularização na programação, que é a divisão do código do programa em sub-rotinas. Assinale a alternativa que contenha tipos de sub-rotinas:**

A alternativa **"C "** está correta.

Na modularização do código de um programa, os tipos de sub-rotinas encontradas são as funções e os procedimentos.

**2. Assinale a alternativa que contenha a afirmativa correta sobre função e procedimento:**

A alternativa **"C "** está correta.

Na modularização do código de um programa, os tipos de sub-rotinas encontradas são as funções e os procedimentos. As funções são um tipo de procedimento que retorna algum valor para quem as chamou, diferente do procedimento, que não retorna nenhum valor.

# MÓDULO 2

---

⦿ **Identificar os tipos de funções predefinidas na linguagem C**

# CONCEITOS

As funções predefinidas na linguagem C estão estipuladas nas bibliotecas da linguagem, nos arquivos header com extensão “.h”. Para usá-las nos programas, é necessário incluir no início

do mesmo, através da diretiva #include, a biblioteca que possui a função que será usada.

Dessa maneira, existem diversos header padrão na linguagem C, entre eles:

| | |
|---|---|
| stdio.h | Funções de E/S padrão nos dispositivos stdin, stdout e files. |
| stdlib.h | Funções de E/S padrão nos dispositivos stdin, stdout e files.Funções para conversão de números em cadeias de caracteres. |
| ctype.h | Funções para tratamento de caractere. |
| time.h | Funções para manipulação de horários e datas. |
| dos.h | Funções de acesso as INT's da BIOS e DOS do MS-DOS. |
| string.h | Funções de manipulação de strings (TC). |
| math.h | Funções matemáticas em geral. |

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

# FUNÇÕES DE E/S PADRÃO

As funções de E/S padrão são responsáveis pelas operações de entrada e saída de dados. Quando um programa na linguagem C é executado, o sistema operacional é responsável por abrir três arquivos: O arquivo de entrada padrão (stdin), o arquivo de saída padrão (stdout) e o erro padrão (stderr).

Os arquivos stdout e stderr são direcionados para a saída do vídeo, e o arquivo stdin é direcionado para o teclado.

# DECLARAÇÃO

Para ter acesso a essas funções, deve-se fazer a seguinte declaração no início do programa:

#include <stdio.h>

# INVOCANDO FUNÇÕES DE E/S PADRÃO

**1) printf**

**Funcionalidade:** Função responsável pela impressão formatada dos dados. Ela imprime os dados no dispositivo de saída padrão, tela do computador. Os mesmos são formatados de acordo com os códigos de formatação, conforme a Tabela 1:

| Caractere | Impresso Como | Tipo de Argumento |
|---|---|---|
| d, i | Número decimal | int |
| o | Número octal sem sinal (sem zero à esquerda) | int |
| x,X | Número hexadecimal sem sinal (sem um 0x ou 0X à esquerda), usando abcdef ou ABCDEF para 10,…, 15 | int |
| U | Número decimal sem sinal | int |

| | | |
|---|---|---|
| C | Único caractere | int |
| S | Imprime caracteres da string até um ‘\ o’ ou o número de caracteres dado pela precisão | char |
| F | [-] m.dddddd, onde o número de d's é dado pela precisão (padrão 6) | double |
| e, g | [-] m.dddddde+/-xx or [-] m.ddddddE+/-xx, onde o número de d's é dado pela precisão (padrão 6) | double |
| g, G | Use % e ou % E se o expoente for menor que = 4 ou maior ou igual à precisão; caso contrário, use %f. | double |
| P | Ponteiro | void * |
| % | Nenhum argumento impresso, imprima um % | |

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Tabela 1: Tabela de conversões para a função printf. Fonte: EnsineMe.

**Sintaxe:**

printf(“format_string”), arg1, arg2, ...);

Onde:

**format_string:** É formado por caracteres ordinários e especificações de formato. Os caracteres ordinários são copiados diretamente na saída padrão. A especificação de formato é composta pelos seguintes elementos:

% [FLAGS] [WIDTH] [.PRECISION] [SIZE] TYPE

# FLAGS:

Composto pelos seguintes caracteres:

"-" → Posicione o valor à esquerda.

"+" → O valor deve ser precedido do sinal ("+" ou "-").

" " → Implica na impressão com sinal negativo apenas.

“0” → Preenche o valor com zeros.

# WIDTH:

Composto pelos seguintes valores:

<n> → Ao menos <n> caracteres são impressos. Caso o comprimento seja menor que <n>, será completado com brancos.

O <n> → Caso o comprimento seja menor que <n>, completa com zeros.

# PRECISION:

Composto pelos seguintes valores:

.0 → Impressão de float sem ponto decimal.

. <n> → Limita o número de casas decimais a <n>.

# SIZE:

Composto pelos seguintes caracteres:

l → Implica na definição de um modificador LONG.

h → Implica na definição de um modificador SHORT.

# TYPE:

Composto pelos seguintes caracteres:

d → O argumento é dado na forma inteiro decimal.

o → O argumento é apresentado em octal (inteiro).

x → O argumento é dado na forma inteiro hexadecimal.

u → O argumento é considerado decimal inteiro sem sinal.

c → O argumento é um caractere único.

s → O argumento é uma cadeia de caracteres (string).

e → O argumento é dado como float ou double em notação científica (com expoente).

f → O argumento é dado como float ou double em notação decimal comum (sem expoente).

g → Usar "E" ou "F", conforme a necessidade.

p → O argumento é um pointer (TC).

% → Reproduz o próprio símbolo "%".

**arg1, arg2, arg3, ... (argumentos):** São apontadores para a área de armazenamento, ou seja, ponteiros para variáveis (fornecem o endereço da variável, exceto para arrays). O primeiro argumento é obrigatório. Se o programador invocar a função printf sem argumentos − printf() − uma mensagem de erro é emitida: ***too few arguments to function 'printf'***.

**Exemplos:**

No comando a seguir, a nota do aluno, que é um valor em ponto flutuante que será exibido com 3 dígitos e 2 casas decimais:

```
printf ("\nNota do aluno:%3.2f", nota);
```

No comando a seguir, será impressa a variável string como uma cadeia de caracteres, e a variável contador será impressa em hexadecimal:

```
printf ("Lista de itens: %s\nContador = %x", string, contador);
```

No comando a seguir, as variáveis a, b e c serão impressas como números decimais, alinhadas à esquerda e com 5 dígitos:

```
printf ("a: %-5D b: %-5D c: %-5D", a, b, c);
```

Analisemos o programa a seguir (Programa1.c):

```
//Programa 1
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
int i;
i=123;
printf("%d\n", i);
return 0;
}
```

O primeiro argumento da função **printf()** corresponde a um arranjo de caracteres (string), que são tratados da seguinte forma:

Caracteres ordinários que são copiados para o fluxo de saída.

Caracteres de especificação de conversões.

A distinção entre esses dois tipos relaciona-se com o caractere ‘%’ (por cento). Os caracteres que seguem o ‘%’ devem obedecer a várias regras e, se estas não forem obedecidas, o comportamento não é definido na linguagem C.

No programa mencionado anteriormente, os caracteres ‘%’ e ‘d’ correspondem a uma especificação de conversão. Umas das regras da função printf() é que, além do primeiro argumento, deve haver tantos argumentos quanto o número de especificações de conversão.

O caractere ‘d’ após o ‘%’, nesse programa, especifica que deve haver um argumento que será convertido como um valor do tipo int na base 10, será impresso 123 na saída padrão e pulada uma linha por causa do /n.

Os caracteres '\' e 'n' correspondem a caracteres ordinários e especificam o envio de um caractere "funcional" correspondente a uma "nova linha".

Substituindo a chamada da função printf("%d\n", i); por printf("%x\n", i); ocorre a especificação de conversão do argumento como um valor do tipo int, mas na base 16, e seria impresso o valor 7B, que corresponde a 123 em decimal.

Analisemos outro programa (Programa2.c):

```
//Programa 2
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
double pi;
pi=3.14;
printf("%f\n", pi);
return 0;
}
```

Esse programa é similar ao anterior, mas agora é utilizada uma variável do tipo double e é especificada uma conversão de um tipo double para o formato [-] m.dddddd, através da especificação de formato %f.

Os dois programas anteriores podem ser combinados, e teremos o Programa3.c:

```
//Programa 3
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
int i; double pi;
i=123; pi=3.14;
printf("%d %f\n", i, pi);
return 0;
}
```

As duas especificações de conversão (%d e %f), exigem dois argumentos (i e pi), que serão impressos respectivamente pelos formatos especificados, decimal e ponto flutuante.

**2) scanf**

**Funcionalidade:** Função responsável pela entrada formatada dos dados. Ela lê os dados do dispositivo de entrada padrão (teclado do computador), que são formatados de acordo com os códigos de formatação conforme a Tabela 2. Pertence ao grupo de funções de entrada formatada, e atua de forma análoga à saída formatada (printf), mas no sentido inverso.

| Caractere | Dados de entrada | Tipo de Argumento |
|---|---|---|
| d | Número inteiro decimal | int * |
| i | Inteiro | int * |
| o | Octal inteiro | int * |
| u | Inteiro decimal sem sinal | unsigned int* |
| x | Número inteiro hexadecimal | int * |
| c | Caracteres | char * |
| s | Cadeia de caracteres (string) | char * |
| e, f, g | Número de ponto flutuante com o ponto decimal opcional e expoente opcional | float * |
| % | % Literal | |

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Tabela 2: Tabela de conversões para a função scanf.

**Sintaxe:**

scanf ("format_string", arg1, arg2, arg3, ...);

Onde:

**format_string:** Opera de forma semelhante ao printf, entretanto, agora serão apresentadas as formatações para a entrada de dados que serão lidos pelo teclado.

**arg1, arg2, arg3, ... (argumentos):** São apontadores para a área de armazenamento, ou seja, ponteiros para variáveis (fornecem o endereço da variável, exceto para arrays), sendo obrigatório o primeiro argumento. Se o programador invocar a função scanf sem argumentos (scanf()), uma mensagem de erro é emitida: ***too few arguments to function 'scanf'***.

# 📢 ATENÇÃO

É importante não se esquecer de colocar o “&” antes do nome da variável. A falta deste caractere é um erro muito comum dos programadores iniciantes

**Exemplos:**
A função scanf a seguir irá ler os dados digitados no teclado e armazenará na variável salario como tipo float:

scanf(“%f”, &salario);

Agora, a função **scanf** irá armazenar os dados digitados no teclado como inteiro na variável número:

scanf("%i", &numero);

Analisemos o programa abaixo (Programa4.c):

```
//Programa 4
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[]){
int i;
scanf("%i", &i);
printf("%i\n", i);
return 0;
}
```

Esse programa usa a função **scanf()** para obter um valor para a variável i a partir dos caracteres que foram digitados no teclado. O dado será armazenado na variável i como um inteiro. A função **printf** irá exibir na tela o conteúdo da variável com no formato inteiro.

A função scanf() lê uma ou mais linhas para obter caracteres a serem usados nas conversões. Um possível problema que pode ocorrer na função scanf() é um certo tratamento desigual para as diferentes conversões.

## COMENTÁRIO

Quando é especificado no primeiro parâmetro uma conversão do tipo %d, a função scanf() irá ignorar diversos caracteres para poder encontrar uma sequência de caracteres entre ‘0’ e ‘9’. Mas, com relação à conversão %c, a função scanf() não ignora nenhum caractere (nem mesmo o correspondente ao momento em que o usuário teclou “enter”).

A função scanf retorna ao ponto de chamada somente após ter processado toda a especificação de conversão dada pelo primeiro argumento. O caractere de mudança de linha é tratado como caractere em branco exceto na conversão %c.

**3) Outras funções para entrada e saída de dados:**

# PUTCHAR

Essa função envia um único caractere pela saída padrão (escreve apenas um caractere).
Equivale ao comando: printf("%c",carac).
Sintaxe: putchar();

# PUTS

Essa função envia uma string na saída padrão. A string será adicionada com um caractere '\n'.
Equivale a: printf("%s\n", string).
Sintaxe: puts(string);

## GETCHAR

Essa função é responsável por ler um caractere da entrada padrão. Equivale à função putchar, mas em sentido inverso.

Sintaxe: [variável] = getchar();

## GETS

Essa função é responsável por ler uma string da entrada padrão. Análoga à função puts, mas em sentido inverso.

Sintaxe: [variável] = gets();

## 📢 ATENÇÃO

Para que nenhum lixo do buffer de teclado atrapalhe o uso da função getchar, coloque fflush(stdin), no Windows, ou _fpurrge(stdin), no Linux, antes da leitura do caractere.

# FUNÇÕES DE MANIPULAÇÃO DE ARQUIVOS

As funções de manipulação de arquivos são responsáveis pelas operações com estes, e se encontram na biblioteca stdio.h

**Declaração**

Para ter acesso a essas funções, deve-se fazer a seguinte declaração no início do programa:

#include <stdio.h>

## INVOCANDO FUNÇÕES DE MANIPULAÇÃO DE ARQUIVOS

**1) fopen**

**Funcionalidade:** A função fopen abre um arquivo, assim como os dispositivos console, impressora e saída serial, que podem ser abertos como arquivos.

As funções de manipulação de arquivos, assim como os ponteiros, devem ser declaradas juntamente com a declaração de variáveis. Sendo assim, vamos entender melhor a definição de ponteiros para arquivos conforme a sintaxe abaixo:

FILE *fp;

**Onde:**

− FILE: Tipo de dado definido no stdio.h.

− *fp: Ponteiro para o arquivo que será manipulado no programa.

**Sintaxe:**

FILE *fopen(const char *nome_arquivo, const char *modo);

**Onde:**
**nome_arquivo** é o nome do arquivo aberto, e o modo pode conter um dos seguintes valores:

"r" − Leitura (open).

"w" − Escrita (rewrite).

"a" − Adição (append).

"r+" − Para atualizar um arquivo (read e write).

"w+" − Para criar um arquivo para atualização.

"a+" − Adição, em arquivo para leitura e escrita.

A função fopen() retorna o ponteiro para o arquivo ou NULL, em caso de erro na abertura do mesmo. Todas as outras operações sobre arquivos serão referenciadas através deste "apontador de arquivo".

Após declarar a variável do tipo arquivo, pode ser aberto o arquivo utilizando a função fopen(), conforme abaixo:

fp = fopen(“arquivo.txt”,”r”);

**2) Outras funções**

# FGETC

Responsável pela leitura de um caractere de um arquivo indicado pelo ponteiro file_pointer. No caso de chegar ao fim do arquivo, será retornado EOF. Sintaxe: [caractere] = fgetc (file_pointer);

# FPUTC

Responsável pela inserção de um caractere no arquivo. A função escreve o caractere na próxima posição do arquivo apontado pelo file_pointer. É a operação inversa de getc.
Sintaxe: fputc (caractere, file_pointer);

# UNGETC

Responsável em "devolver" o último caractere lido do arquivo. A função devolverá o caractere dado para o arquivo apontado pelo file_pointer. Apenas um caractere devolvido será aceito (coloca em um buffer temporário de um caractere) por cada arquivo.
Sintaxe: ungetc (caractere, file_pointer);

# FPRINTF

Responsável pela saída formatada em arquivos. É equivalente à função printf, mas com a diferença de que esta atua sobre arquivos, os quais são descritos através do primeiro parâmetro, que é um ponteiro para o arquivo sobre o qual se deseja escrever.
Sintaxe: fprintf (file_pointer,"controle", arg1, arg2,...);

# FSCANF

Responsável pela entrada formatada a partir de arquivos. Atua como a função inversa de fprintf e análoga à função scanf, só que atuando sobre arquivos.
Sintaxe: fscanf (file_pointer,"controle",arg1,arg2,...);

# FCLOSE

Responsável pelo fechamento do arquivo em uso.
Sintaxe: fclose (file_pointer);

# OUTRAS FUNÇÕES

| Quadro Resumo | |
|---|---|
| | |

| | |
|---|---|
| fflush() | Descarrega o buffer. |
| fgets() | Obtém uma string do arquivo. |
| fputs() | Insere uma string no arquivo. |
| fread() | Lê um bloco de dados do arquivo. |
| fwrite() | Escreve um bloco de dados no arquivo. |
| fseek() | Reposiciona o ponteiro para o arquivo. |
| rewind() | Reposiciona o ponteiro para o início do arquivo. |
| ftell() | Retorna a posição do ponteiro. |

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

# FUNÇÕES DE MANIPULAÇÃO DE TIPOS DE DADOS

As funções de manipulação de tipos de dados podem ser usadas para tratamento de caractere, permitindo verificar se um determinado caractere é ASCII, se é numérico, se é uma letra maiúscula, minúscula etc.

# DECLARAÇÃO

Para ter acesso a essas funções, deve-se fazer a seguinte declaração no início do programa:

#include <ctype.h>

# INVOCANDO FUNÇÕES DE MANIPULAÇÃO DE TIPOS DE DADOS

# ISUPPER

Verifica se o caractere é maiúsculo. Retorna diferente de zero se for maiúsculo e zero no caso contrário (minúscula). Possui função análoga, mas de funcionamento inverso a esta: islower (c), que verifica se o caractere é minúsculo.
Sintaxe: valor_log = isupper (caractere);

# ISALPHA

Verifica se é caractere alfabético. Segue o mesmo princípio de isupper.
Sintaxe: valor_log = isalpha (caractere);

Funções análogas:

isdigit (c) − Verifica se é um dígito.

isspace (c) − Verifica se é caractere de espaço.

isascii (c) − Verifica se é caractere ASCII.

isprint (c) − Verifica se é caractere de impressão.

# TOLOWER

onverte de maiúscula para minúscula. Possui função análoga, mas de funcionamento inverso a esta: toupper (c).
Sintaxe: caractere = tolower (caractere);

# FUNÇÕES DE MANIPULAÇÃO DE STRING

As funções de manipulação de strings podem ser usadas para copiar, comparar, concatenar, dentre outras funcionalidades.

## DECLARAÇÃO

Para ter acesso a essas funções, deve-se fazer a seguinte declaração no início do programa:

#include <string.h>

## INVOCANDO FUNÇÕES DE MANIPULAÇÃO DE STRING

## STRLEN

Retorna o tamanho de uma string.

Sintaxe: strlen(string); onde "string" é um tipo char *c.

## STRCPY

Copia uma string em outra.

Sintaxe: strcpy(s,t); onde "s" e "t" são do tipo char *c.

## STRCMP

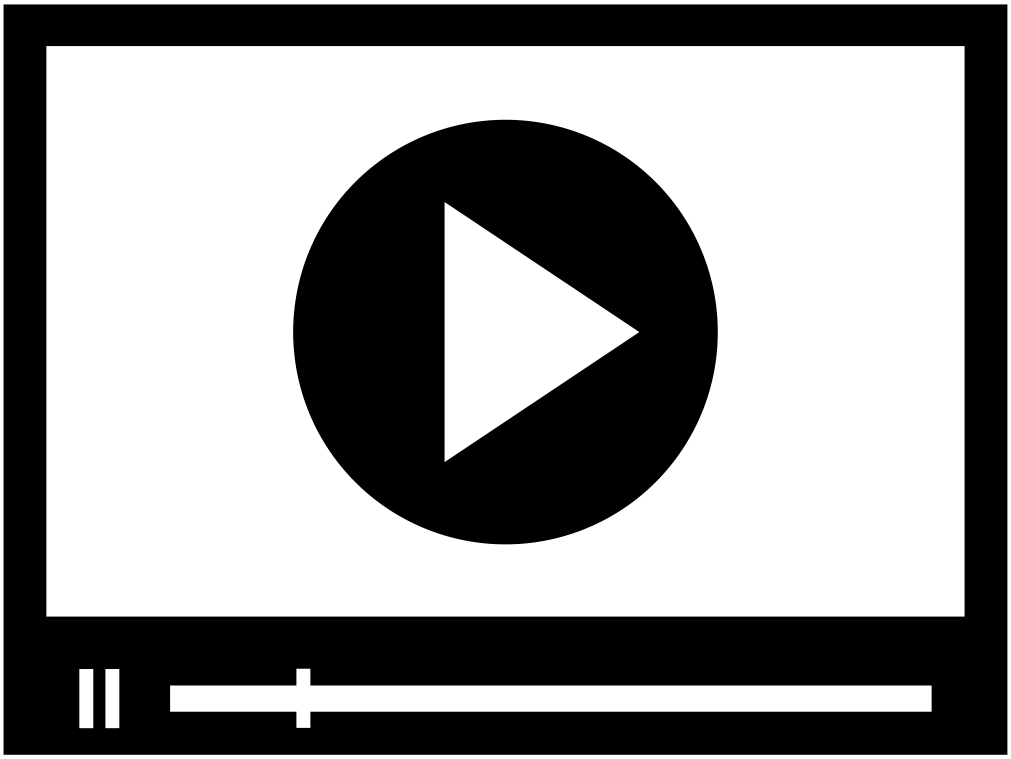
Compara duas strings.

Sintaxe: strcmp(s,t); onde "s" e "t" são do tipo char *c.

## STRCAT

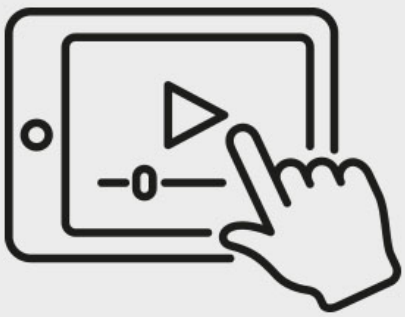
Concatena duas strings.

Sintaxe: strcat(s,t); onde "s" e "t" são do tipo CHAR *c.



# EMPREGANDO FUNÇÕES PREDEFINIDAS

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



# VERIFICANDO O APRENDIZADO

**1. NA LINGUAGEM DE PROGRAMAÇÃO C, AS FUNÇÕES PREDEFINIDAS ESTÃO NAS BIBLIOTECAS DA LINGUAGEM. CADA BIBLIOTECA-PADRÃO TEM UM CABEÇALHO QUE CONTÉM OS PROTÓTIPOS PARA**

**TODAS AS FUNÇÕES, ASSIM COMO DEFINIÇÕES DE VÁRIOS TIPOS DE DADOS E CONSTANTES QUE SÃO NECESSÁRIAS PARA AS MESMAS. UMA DESSAS BIBLIOTECAS TEM A SEGUINTE EXPLICAÇÃO: FUNÇÕES DE ENTRADA E SAÍDA PADRÃO. ASSINALE-A:**

**A)** stdio.h

**B)** stdlib.h

**C)** ctype.h

**D)** dos.h

**E)** string.h

**2. CONSIDERE O PROGRAMA C, LISTADO A SEGUIR.**

```
#INCLUDE <STDIO.H>
INT MAIN(VOID) {
INT A, *Y, Z;
X = 5;
Y = &X;
Z = 10;
SCANF("%D", Y);
PRINTF("%D %D ", X, Z);
}
```

**ASSINALE A ALTERNATIVA QUE REPRESENTA O QUE SERÁ IMPRESSO PELO PROGRAMA SE O USUÁRIO DIGITAR 15, COMO ENTRADA DE DADOS:**

**A)** 10 15

**B)** 15 10

**C)** 5 10

**D)** 5 15

**E)** Null pointer

---

## GABARITO

**1. Na linguagem de programação C, as funções predefinidas estão nas bibliotecas da linguagem. Cada biblioteca-padrão tem um cabeçalho que contém os protótipos para todas as funções, assim como definições de vários tipos de dados e constantes que são necessárias para as mesmas. Uma dessas bibliotecas tem a seguinte explicação: Funções de entrada e saída padrão. Assinale-a:**

A alternativa **"A "** está correta.

A biblioteca stdio.h contém várias funções de entrada e saída, como : scanf, printf, getc, gets, putc, puts, entre outras funções.

**2. Considere o Programa C, listado a seguir.**

```
#include <stdio.h>
int main(void) {
int a, *y, z;
x = 5;
y = &x;
z = 10;
scanf("%d", y);
printf("%d %d ", x, z);
}
```

**Assinale a alternativa que representa o que será impresso pelo programa se o usuário digitar 15, como entrada de dados:**

A alternativa **"B "** está correta.

Observe no código que o valor digitado será armazenado na variável y, que é um ponteiro para a variável x. Sendo assim, o valor impresso na variável x é 15. Como a variável z recebe no código o valor 10, será impresso esse valor.

# MÓDULO 3

---

⦿ **Distinguir passagem de parâmetros por valor e por referência**

# INTRODUÇÃO

No módulo anterior, estudamos as funções predefinidas, tais como printf, scanf, getchar, putchar, dentre outras que fazem parte da biblioteca padrão C, ou seja, são obtidas ao se adquirir qualquer compilador de C tal como GCC.

Essas funções, de acordo com a sua funcionalidade, possuem parâmetros, e são invocadas pelo programa principal passando argumentos de acordo com a sua sintaxe.

## ★ EXEMPLO

Por exemplo, a função scanf é responsável por efetuar a leitura dos dados de uma fonte externa. E, ao analisar sua sintaxe − scanf(“expressão de controle”, lista de argumentos) − observamos que possui argumentos que declaram o formato dos valores a serem lidos e os endereços das variáveis que irão receber esses valores da fonte externa.

Sendo assim, vamos entender a passagem desses valores entre o programa principal e suas sub-rotinas.

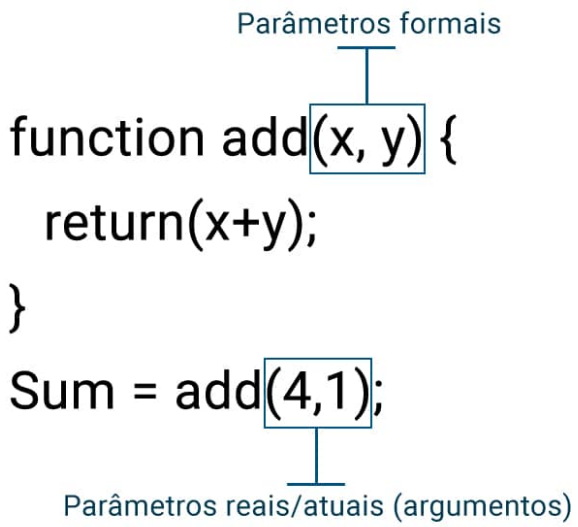
# DEFINIÇÕES

Na computação, um parâmetro é uma variável que recebe valores passados para sua sub-rotina. Esta utiliza esses valores atribuídos aos parâmetros para alterar o seu comportamento em tempo de execução.

Parametriza-se, portanto, a sub-rotina com o objetivo de receber informações que virão posteriormente, durante a execução do programa.

Os parâmetros, também chamados de parâmetros formais, são definidos na declaração da sub-rotina. O nome de cada um deles serve como uma variável local dentro da função. Os argumentos, também conhecidos por parâmetros reais/atuais, representam os valores, ou seja, as variáveis atuais passadas para os parâmetros da sub-rotina, quando é invocada.

O exemplo a seguir demonstra essas definições: Considere uma função que realiza a operação de adição de dois números inteiros. Ela (function add) possui dois parâmetros: x e y, que são os parâmetros formais. Ao ser invocada a função add através da linha de comando sum = add(4,1); são passados dois argumentos, que são os números 4 e 1.




Fonte: EnsineMe

Exemplo de Definição. Fonte: EnsineMe.

# PASSAGEM DE PARÂMETROS

Quando invocamos uma sub-rotina, devemos fornecer para cada um dos seus parâmetros um valor de mesmo tipo, respeitando a ordem e a quantidade de parâmetros declarados. Essa passagem pode ser feita de duas formas:

Passagem por valor

A primeira é através de uma cópia do valor do argumento na chamada da sub-rotina para o parâmetro declarado na sub-rotina, que se denomina **passagem por valor**.

Passagem por referência

A segunda maneira é através da passagem do endereço onde se encontra a variável usada como argumento na chamada da sub-rotina, conhecida por **passagem por referência**.

Para ilustrar os dois tipos de passagem, vamos considerar a troca de valores entre variáveis, que é um problema frequente de programação e pode ser usado na ordenação de um vetor, por exemplo. Uma troca é realizada em três passos e sempre com o auxílio de uma variável auxiliar.

O Programa 5, a seguir, cria uma função troca() que realiza a troca de valores entre duas variáveis:

```
//Programa 5
#include <stdio.h>
void troca(int x, int y) {
int aux;
aux = x;
x = y;
y = aux;
}
int main(){
int v1=5, v2=10;
printf("Antes da troca: v1=%d e v2=%d\n", v1, v2);
troca(v1,v2);
printf("Depois da troca: v1=%d e v2=%d\n", v1, v2);
exit(0);
}
```

O programa principal invoca a sub-rotina troca() com dois argumentos, a variável v1 e a variável v2, na linha 12.

```
troca(v1,v2);
```

Os parâmetros da sub-rotina troca(), x e y, recebem os valores 5 e 10, respectivamente. Esses parâmetros são as variáveis da sub-rotina.

A sub-rotina troca() possui também uma variável “aux”, que recebe o valor do parâmetro x. Em seguida, a variável x recebe o valor da variável y, e a variável y recebe o valor de aux.

Sendo assim, ao executar o programa:

O programa principal declara duas variáveis inteiras, v1 e v2, que recebem os valores 5 e 10, respectivamente.

O programa principal invoca a sub-rotina troca() com dois argumentos, o valor da variável v1 e o valor da variável v2 - troca(5, 10).

A sub-rotina troca() recebe os valores nos seus parâmetros, x e y - troca(5,10).

A sub-rotina executa os comandos: Variável local aux recebe o valor do parâmetro x, x recebe o valor do parâmetro y, e y recebe o valor da variável aux, ficando as variáveis com os seguintes valores: aux = 5, x = 10 e y = 5.

A troca de valores, porém, não foi realizada na função main, apenas na função troca.

Ao invocarmos uma sub-rotina passando variáveis no lugar dos parâmetros, os valores das mesmas são copiados para os parâmetros da sub-rotina. Sendo assim, as alterações dentro da sub-rotina no valor dos parâmetros não afetam as variáveis usadas na chamada desta.

Essa forma de relacionamento do programa principal com a sub-rotina chamamos de **passagem por valor** . Nela, é passada realmente uma cópia do valor indicado na chamada do procedimento ou função. Se alterarmos o conteúdo da variável recebida como parâmetro, alteraremos a cópia do valor original e, portanto, o original não é modificado.

Podemos dizer que a passagem por valor é feita entregando a “xerox” das informações, jamais o original. Dessa forma, a sub-rotina pode “riscar e rabiscar” os dados que foram passados, sem, no entanto, danificar o “documento original”.

Voltando ao exemplo anterior, quando o programa principal é executado, são criadas as variáveis v1 e v2. Por terem sido declaradas na função principal do programa, sua visibilidade (ou seja, o escopo para se manipulá-las) é restrita a esta função.

Quando a função troca() é invocada, novas variáveis x e y, visíveis apenas dentro da função troca(), são criadas, suspendendo temporariamente a visibilidade de v1 e v2 criadas em main().

A chamada de troca() em main(), porém, resulta na passagem da cópia dos valores de v1 e v2 de main() para as novas variáveis x e y que são criadas em troca().

Assim, dentro da função troca(), quaisquer alterações de x e y não resultarão em modificações de v1 e v2 em main(), pois o que está sendo alterado é apenas uma cópia. Ou seja, caracterizou a passagem por valor.

## COMENTÁRIO

Mas quando desejamos passar uma variável para uma sub-rotina de modo que seu valor possa ser alterado pela sub-rotina, devemos utilizar a **passagem por referência.**

Nesse caso, o parâmetro da sub-rotina deve ser declarado como ponteiro para um tipo. E na chamada da sub-rotina deve-se passar o endereço de memória da variável, de mesmo tipo do ponteiro, que terá seu valor alterado.

O artifício corresponde em passar como argumento para a sub-rotina o endereço de memória da variável, não o seu valor. Portanto, é enviada para a função não uma cópia do valor da variável, e sim uma referência a esta. Assim, qualquer alteração dos parâmetros realizada na função corresponde a uma alteração nas variáveis originais.

Então, para realizar a passagem por referência, ponteiros devem ser utilizados. A estratégia a ser adotada é:

# PASSAR O ENDEREÇO DE UMA VARIÁVEL, OBTIDO ATRAVÉS DO OPERADOR & E ARMAZENADO EM UMA VARIÁVEL DO TIPO PONTEIRO.

# DENTRO DA FUNÇÃO, USAR PONTEIROS PARA ALTERAR OS LOCAIS PARA ONDE ELES APONTAM E QUE CORRESPONDEM AOS VALORES DAS VARIÁVEIS ORIGINAIS.

Dessa forma, podemos alterar o conteúdo da variável pela passagem por referência.

Vamos voltar ao Programa 5, para reescrevê-lo e alterar o valor das variáveis. Chamaremos de Programa 6:

```
//Programa 6
#include <stdio.h>
void troca(int *x, int *y) {
int aux;
if(x != NULL && y != NULL){ //se endereços forem válidos
aux = *x; //faz a troca
*x = *y;
*y = aux;
}
}

int main(){
int v1=5, v2=10;
troca(&v1, &v2);
printf("v1 = %d e v2 = %d\n", v1, v2);
}
```

Note que a variável aux continua a ser declarada do tipo int, pois seu propósito é armazenar um dos valores e não endereço.

A sub-rotina troca() é invocada, e nos seus argumentos, no lugar dos valores das variáveis v1 e v2, encontramos o endereço de memória das variáveis ao se usar o operador “&”.

```
troca(&v1, &v2);
```

A sub-rotina troca() possui nos seus parâmetros os ponteiros *x e *y, responsáveis em receber o endereço de memória das variáveis v1 e v2.

```
void troca(int *x, int *y)
```

Sendo assim, ao executar o programa:

O programa principal declara as variáveis x e y, que recebem os valores 5 e 10, respectivamente (x=5 e y=10).

O programa principal invoca a sub-rotina troca(), que passa por referência o endereço de memória das variáveis v1 e v2, por exemplo: troca(6487580, 6487576).

A sub-rotina troca() recebe os endereços de memória das variáveis v1 e v2 nos seus parâmetros *x e *y (void troca(6487580, 6487576)).

Os ponteiros *x e *y, parâmetros da sub-rotina, agora estão com os endereços das variáveis v1 e v2.

A variável aux, declarada localmente na sub-rotina, recebe o valor da variável v1, apontada pelo ponteiro *x (aux = 5).

O valor armazenado no endereço de memória apontado por *x é trocado pelo valor armazenado no endereço de memória apontado pelo *y.

O valor armazenado no endereço de memória apontado por *y é trocado pelo valor da variável aux.

Na passagem por referência, se alterarmos o conteúdo da variável recebida como parâmetro, alteraremos o valor original, pois temos à nossa disposição o endereço exato onde está localizada na memória e, portanto, podemos alterar diretamente o valor original.

Quando usamos a passagem por referência, não podemos passar valores numéricos ou expressões na chamada da função. Deve ser uma variável única onde possamos definir exatamente o seu endereço.

Conforme mencionado, todo parâmetro das sub-rotinas é destruído ao final da execução da mesma. Isso também é válido para a passagem por referência, mas na realidade o que é destruído é a referência à variável usada no programa principal, e não a própria variável.

# PROTÓTIPOS DE SUB-ROTINAS

Na passagem por valor ou por referência, é necessária a implementação de sub-rotinas no programa principal e, para implementá-la sem partes distintas do arquivo-fonte e depois de utilizá-las, usamos protótipos ou assinaturas de sub-rotina.

Os protótipos ou assinaturas de sub-rotinas correspondem à primeira linha da definição de uma função ou procedimento. Ela deve aparecer antes do uso dessa sub-rotina.

Em geral, colocam-se os protótipos no início do arquivo-fonte. Observe a sintaxe logo abaixo de duas sub-rotinas:

```
tipo nome da função (tipo parâmetro1, ..., tipo parâmetro) {
comandos;
return;
}

void nome do procedimento (tipo parâmetro1, …, tipo parametron) {
commandos;
}
```

A primeira linha das sub-rotinas tipo nome da função (tipo parâmetro1, ..., tipo parâmetron) e void nome do procedimento (tipo parâmetro1, …, tipo parametron) é a assinatura da função e do procedimento, respectivamente. E deve ser inserida no início do programa ou antes do uso da sub-rotina, como apresentado no Programa 7:

```
//Programa 7
#include<stdio.h>;

int soma(int x, int y);//assinatura da função

int main(void) {
int num1, num2, num3;
printf("Digite primeiro numero:");
scanf("%d", &num1);
printf("Digite segundo numero:");
scanf("%d", &num2);
num3 = soma(num1, num2); // Chamada da função
printf("A soma de: %d + %d = %d\n", num1, num2, num3);
return(0);
```

```
}

int soma(int x, int y){ // Definição da função
return (x+y);
}
```

Na linha 4, temos a referência à assinatura da função soma e, na linha 18, a sua definição. Observe que o programa principal invoca a função soma na linha 12, antes da sua definição, e isso somente é possível porque foi inserida a assinatura no início do programa.

# PASSAGEM DE VETORES

A passagem de vetores para funções é sempre por referência. A sintaxe de passagem de um vetor pode ser feita com:

tipo nome[ ];

Onde: Tipo corresponde ao tipo dos elementos do vetor, nome é o nome atribuído ao vetor e [ ] indica que a variável é do tipo vetor.

O “[ ]” pode ser utilizado sem um valor, pois em C não interessa qual a dimensão do vetor que é passado a uma função, mas, sim, o tipo dos seus elementos.

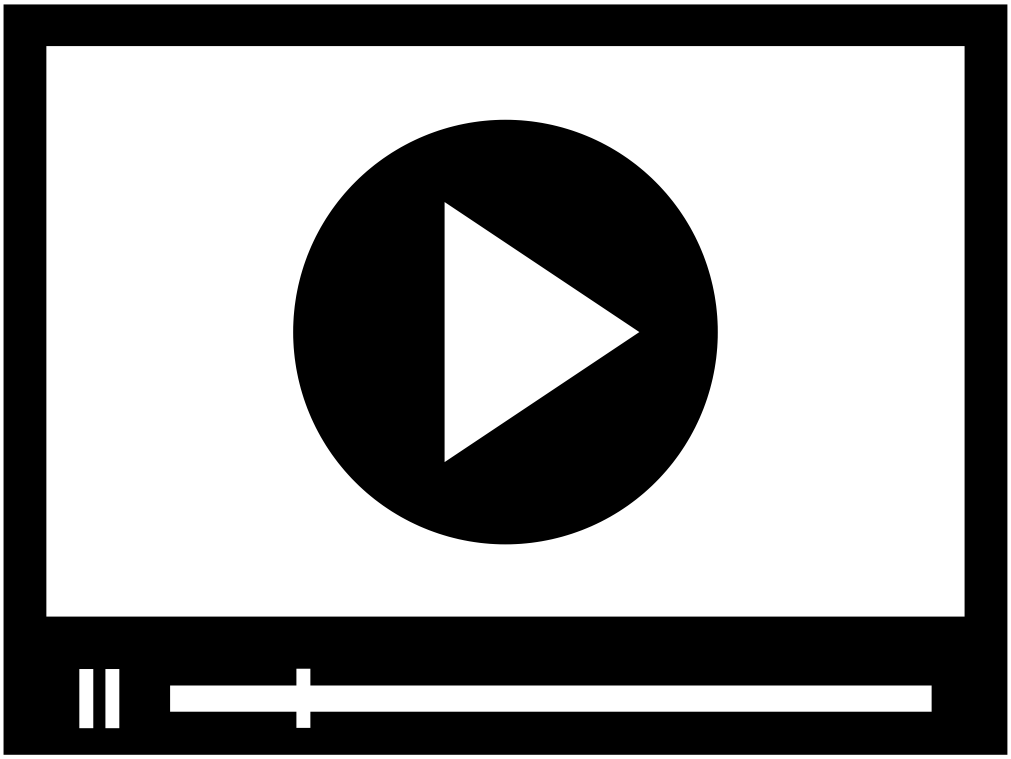
## SAIBA MAIS

Para entender melhor a definição, observe o Programa 8. Ele declara uma função v_iniciacao(), que inicializa o vetor de inteiros com zero. Em seguida, o vetor de 10 posições é alimentado por números inteiros de 0 a 9 e, por último, a função v_imprime() imprime os valores do vetor.

Para passar um vetor para uma função usando linguagem C, utiliza-se um ponteiro como parâmetro da função. Ao usarmos o ponteiro como parâmetro, na realidade estamos passando o endereço inicial do vetor, não os seus elementos.

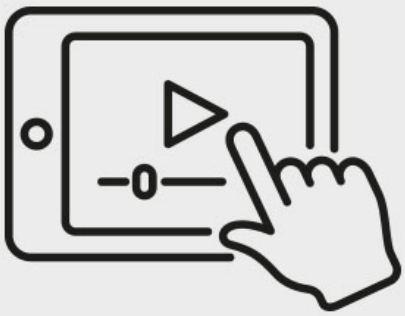
```
//Programa 8
#include <stdio.h>
```

```
void v_iniciacao(int *vet, int n) {
int i;
for (i=0; i<n; i++)
vet[i]=0;
}
void v_imprime(int *vet, int n){
int I;
for(i=0; i < n; i++)
printf(" %d - ",vet[i]);
printf("\n");
}

int main(void) {
innt vet[10], i;
v_iniciacao(vet,10);
// Impressao apos inicializacao.
printf("Impressao do vetor antes da atribuicao.");
v_imprime(vet,10);
// Mudando valores de vet.
for(i=0; i<10; i++){
vet[i]=i;
}
// Impressao apos atribuicao.
printf("Impressao do vetor apos a atribuicao.");
v_imprime(vet,10);
return(0);
}
```



## ENTENDENDO A PASSAGEM DE PARÂMETROS

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## VERIFICANDO O APRENDIZADO

**1. NA PROGRAMAÇÃO EM MÓDULOS, O PROGRAMA PRINCIPAL SE COMUNICA COM AS SUB-ROTINAS ATRAVÉS DE PASSAGEM DE PARÂMETRO. SOBRE OS TIPOS DE PASSAGEM, ASSINALE A SENTENÇA CORRETA:**

**A)** Os parâmetros da função na sua declaração são chamados de variáveis globais.

**B)** Os parâmetros atuais ou reais são os argumentos da função.

**C)** Na passagem por valor, o parâmetro real é passado para o parâmetro formal com o uso dos ponteiros.

**D)** Na passagem por referência é realizada uma cópia do valor a ser passado.

**E)** Tanto na passagem por valor como por referência, os tipos dos parâmetros podem ser distintos.

**2. ANALISE O SEGUINTE CÓDIGO IMPLEMENTADO NA LINGUAGEM C COM PASSAGEM DE PARÂMETRO POR REFERÊNCIA ENTRE A FUNÇÃO MAIN E A FUNÇÃO SOMA.**

```
INT SOMA(INT *A, INT *B) {
*A = *A + *B;
RETURN *A;
}
INT MAIN() {
INT X=5, Y=3;
Y = SOMA(X, Y);
PRINTF("%D", X+Y);
RETURN(0);
}
```

**ASSINALE A OPÇÃO QUE CORRIGE O CÓDIGO:**

**A)** Linha 1: int soma(int a, int b) {

**B)** Linha 8: printf(“%d”, y);

**C)** Linha 6: int x=5; int y=3;

**D)** Linha 2: a = *a + *b;

**E)** Linha 7 y = soma(&x, &y);

---

## GABARITO

**1. Na programação em módulos, o programa principal se comunica com as sub-rotinas através de passagem de parâmetro. Sobre os tipos de passagem, assinale a sentença correta:**

A alternativa **"B "** está correta.

Os parâmetros são passados para uma função de acordo com a sua posição. Na chamada da função, os parâmetros são chamados parâmetros atuais/reais ou argumentos.

**2. Analise o seguinte código implementado na linguagem C com passagem de parâmetro por referência entre a função main e a função soma.**

```
int soma(int *a, int *b) {
*a = *a + *b;
return *a;
}
int main() {
int x=5, y=3;
y = soma(x, y);
printf(“%d”, x+y);
return(0);
}
```

**Assinale a opção que corrige o código:**

A alternativa **"E "** está correta.

A passagem por referência passa o endereço de memória do argumento. Nesse caso, usa-se o operador & para referenciar o endereço de memória da variável.

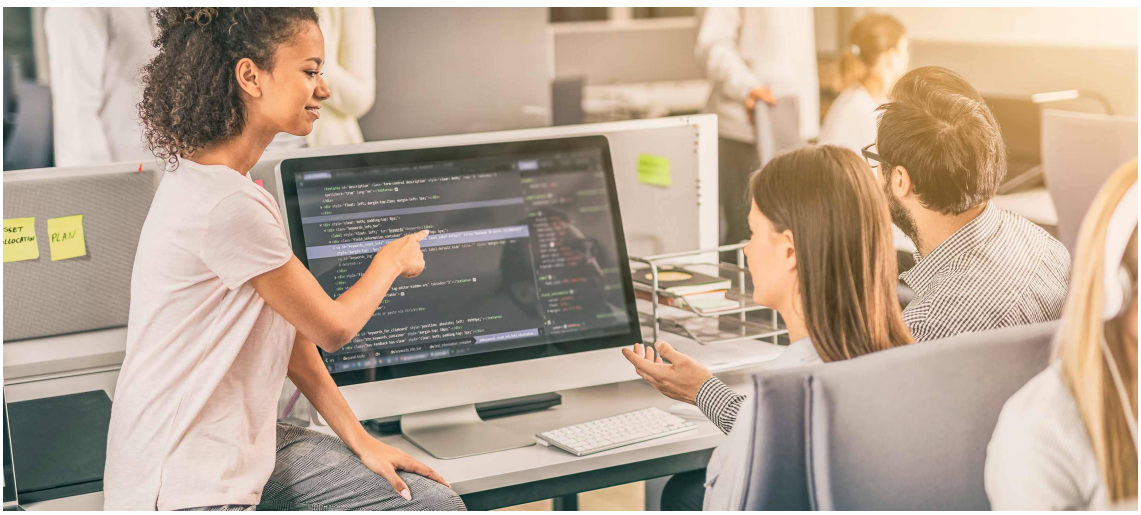
# MÓDULO 4

---

- **Localizar o escopo das variáveis locais e globais**

# INTRODUÇÃO

Nas linguagens de programação, as variáveis são vinculadas aos seus valores em tempo de execução. E o escopo de uma vinculação é o conjunto de trechos de um programa que se consegue usar as variáveis.



No caso da linguagem c, as variáveis são divididas quanto ao escopo em três tipos:

**Variáveis locais.**

**Variáveis globais.**

**Parâmetros formais.**

Sendo assim, podemos declarar as variáveis em três locais distintos:

Primeiro Local

• O **primeiro local** fica fora das sub-rotinas de um programa. Essas são chamadas de **variáveis globais** e podem ser usadas a partir de qualquer lugar do programa. Como elas estão fora das sub-rotinas, podemos dizer que todas as funções as veem.

Segundo Local

• O **segundo local** já fica dentro da sub-rotina de um programa. Essas são chamadas de **variáveis locais** e só têm validade dentro da sub-rotina no qual são declaradas, isto é, só a sub-rotina em que a variável é declarada sabe da sua existência.

Terceiro Local

• O **terceiro local** é na lista de parâmetros de uma sub-rotina. Essas são chamadas de **parâmetros formais** e são declaradas na definição dos parâmetros de uma sub-rotina. Apesar de receberem valores externos, elas são conhecidas apenas pela sub-rotina onde são declaradas.

# VARIÁVEIS LOCAIS

As variáveis locais são declaradas dentro de um bloco de código de uma sub-rotina. Só podem ser usadas ou modificadas pela sub-rotina onde estão declaradas, isto é, dentro desta ou do bloco de código do programa.

## ❓ VOCÊ SABIA

A existência dessas variáveis depende da execução da sub-rotina onde estão declaradas, isso significa que somente existem enquanto a sub-rotina onde foi declarada estiver sendo executada.

Conforme visualizado no Programa 9, a seguir, na linha 5 foram declaradas na função principal main()três variáveis locais: num1, num2 e num3, isto é, essas três variáveis pertencem à função principal main() e seu escopo está relacionado à execução da função.

```
//Programa 9 - soma dois numeros
#include >stdio.h>
```

```
int main ()
{
int num1, num2, num3;
num1 = 10;
num2 = 20;
num3 = num1 + num2;
printf("%d + %d = %d", num1, num2, num3);
return(0);
}
```

Qualquer valor que as variáveis num1, num2 e num3 assumam será válido apenas dentro da função main. Supondo que existisse uma sub-rotina nesse programa, essas variáveis não poderiam ser acessadas.

# VARIÁVEIS GLOBAIS

As variáveis globais são declaradas fora da sub-rotina. São acessíveis em todos os escopos, em qualquer ponto de um programa, , mesmo em outros módulos.

Podem, portanto, ser usadas ou modificadas por qualquer sub-rotina do programa onde estão declaradas. Em outras palavras, as variáveis globais estão disponíveis durante toda a execução do programa.

O valor de uma variável global ao longo do ciclo de vida do programa é sempre válido.

Conforme visualizado no Programa 10, na linha 3 foi declarada a variável num3 como uma variável global:

```
//Programa 10 - soma dois numeros
#include <stdio.h>
int num3;
int main ()
{
int num1, num2;
num1 = 10;;
num2 = 20;
num3 = num1 + num2;
```

```
printf("%d + %d = %d", num1, num2, num3);
return(0);
}
```

Em um programa, as variáveis locais e as globais podem ter o mesmo identificador, isto é, o mesmo nome. Porém, dentro de uma sub-rotina, as variáveis locais sobrepõem o valor das globais.

No Programa 11, apresentado a seguir, encontramos uma declaração de variável global e variável local com o mesmo nome nas linhas 3 e 7, respectivamente, que é a variável num.

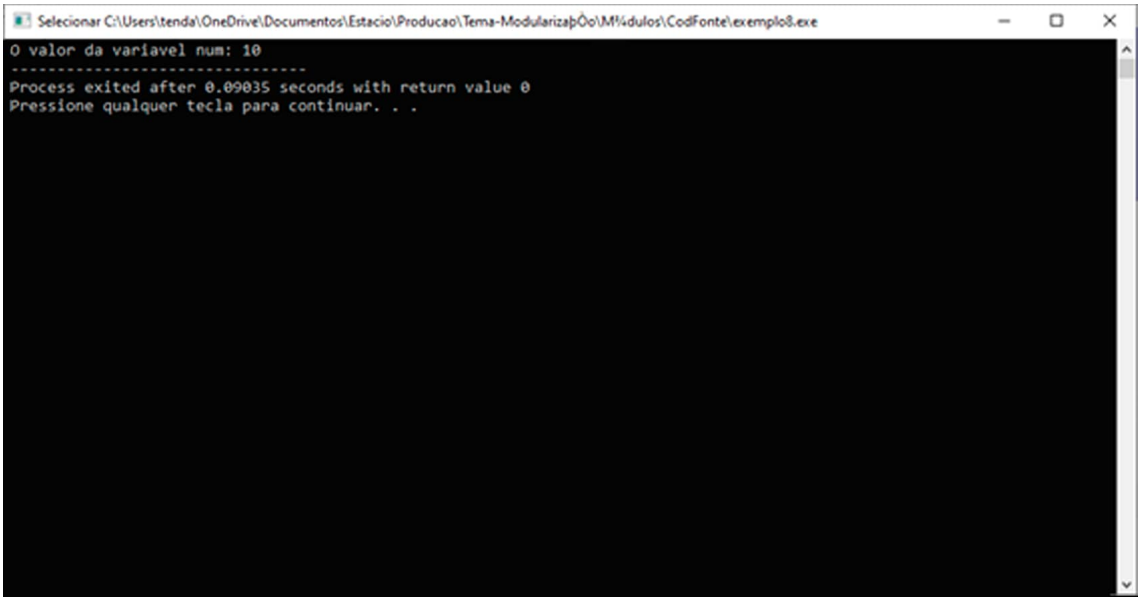
## ★ EXEMPLO

Para exemplificar o que acontece nesse caso, repare que na linha 4 a variável global num recebe o valor 20. Esse valor é válido apenas externamente à função main. Como nesta existe uma variável com o mesmo nome, quando, na linha 8, a variável num recebe o valor 10, esta é a mais interna ao programa, ou seja, a variável local que foi definida na linha7.

Para comprovar, vamos executar o Programa 11. Podemos observar que, ao executar a função printf na linha 9, o valor impresso será o valor 10, que é o atribuído a variável local num:

```
//Programa 11 - imprime o valor da variável num
.#include <stdio.h>
int num;
num = 20;
int main ()
{
int num;
num = 10;
printf("O valor da variavel num: %d", num);
return 0;
}
```

Na figura 1, podemos observar a saída da execução do programa 11:




Fonte: EnsineMe.

Figura 1: Execução do Programa 11.

# PARÂMETROS FORMAIS

Os parâmetros formais de uma sub-rotina também são variáveis locais. Eles são declarados como sendo as entradas de uma sub-rotina, sendo assim, são variáveis locais da função. É possível alterar o valor de um parâmetro formal, pois essa alteração não terá efeito na variável que foi passada à sub-rotina.

Na linguagem C, quando se passa parâmetros para uma sub-rotina, são passadas apenas cópias das variáveis. Isso significa que os parâmetros formais existem independentemente das variáveis que foram passadas para a sub-rotina, eles tomam apenas uma cópia dos valores passados.

## COMENTÁRIO

Observando o Programa 12, a seguir, na linha 7 foi declarada a função soma com dois parâmetros formais, x e y. Essa função recebe dois números inteiros e retorna a soma desses dois números.

Na linha 18, a variável local num3 recebe o retorno da função soma.

Os parâmetros formais da função soma() são variáveis locais dessa função, seus valores são visíveis somente durante a execução da função.

```
//Programa 12: soma dois numeros
#include <stdio.h>
//variável global
int num3;

//Função soma
int soma(int x, int y){
int v_soma;
v_soma = x + y;
return(v_soma);
}

int main ()
{
int num1, num2;
num1 = 10;
num2 = 20;
num3=soma(num1, num2);
printf("%d + %d = %d", num1, num2, num3);
return 0;
}
```

Nesse exemplo, podemos observar ainda sobre variáveis globais: Repare que na linha 4 foi definida a variável global num3. Agora considere que também tenha sido declarada uma variável global num1. Como visto anteriormente, como também há a num1 na função main, a que será manipulada dentro da função main é a variável local declarada na linha 15, portanto, assumindo o valor 10.

E o que aconteceria se na função soma fosse utilizada a variável num1? Qual das duas variáveis num1 seria acessada? Claro que seria a variável global num1, que tem escopo em qualquer parte do programa.

# ESCOPO DE VARIÁVEIS

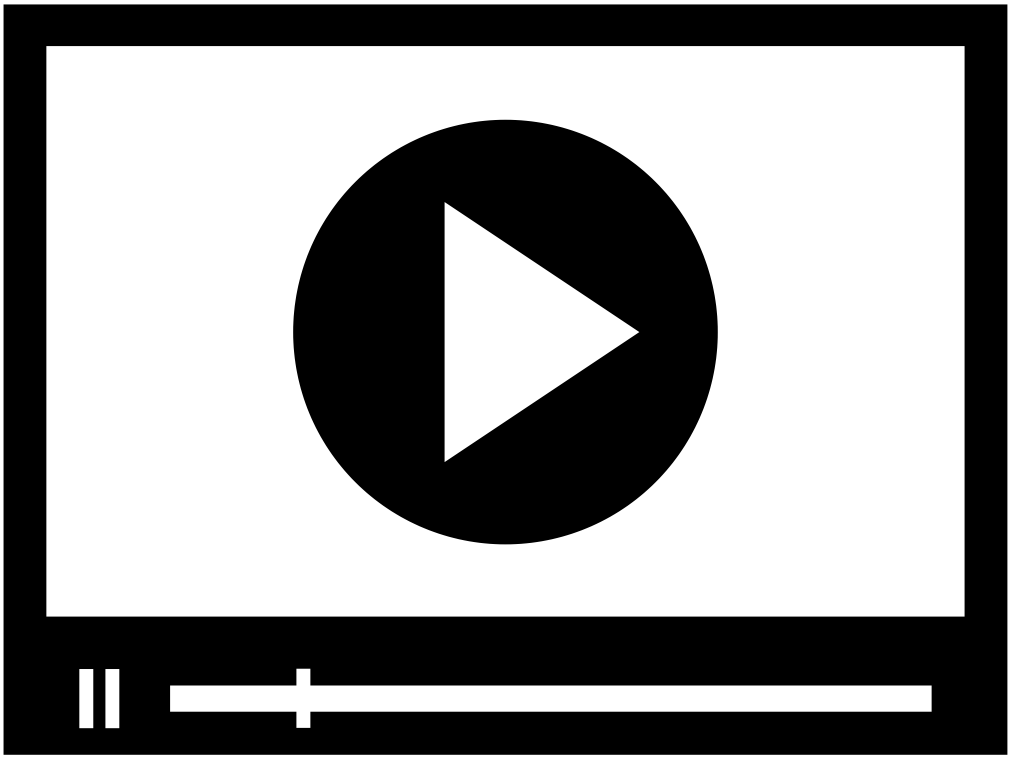
Como visto anteriormente, a declaração de variáveis no programa tem seu escopo vinculado à sua localização no código. Sendo assim, vamos observar o Programa 13 e identificar as variáveis locais, globais e os parâmetros formais:

```
//Programa 13 – Declaração de variáveis locais, globais e parâmetros formais
#include <stdio.h>
#include <conio.h>

//declaração de variáveis globais

void funcao1(variáveis locais de parâmetros)
{
// declaração das variáveis locais da função1
return;
}

// ---------------- Funçao main()-------------------

int main(void)
{
//declaração das variáveis locais da main()
return(0);
}
// ------------------------------------------------
```

As variáveis globais serão definidas fora da função main, ou seja, após a declaração das bibliotecas, e são acessíveis em qualquer parte do programa.

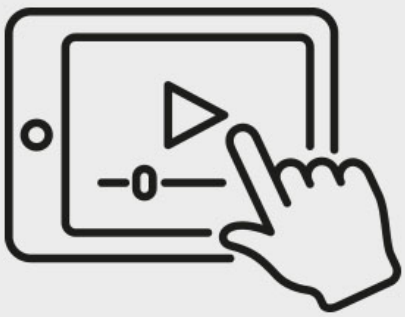
As variáveis locais serão definidas internamente a cada função. No Programa 13, teremos as variáveis locais da função main e da funcao1, que serão válidas apenas dentro de cada função. Caso o valor de uma variável local precise ser utilizado em outra função, esses valores devem ser passados por parâmetros.

Os parâmetros formais estão definidos nos parâmetros das funções e são considerados variáveis locais, portanto, a lista de parâmetros formais estará definida na funcao1().



# ENTENDENDO O ESCOPO DE VARIÁVEIS

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



# VERIFICANDO O APRENDIZADO

**1. NAS LINGUAGENS DE PROGRAMAÇÃO, AS VARIÁVEIS SÃO VINCULADAS AOS SEUS VALORES EM TEMPO DE EXECUÇÃO. E O ESCOPO DE UMA VINCULAÇÃO É O CONJUNTO DE TRECHOS DE UM**

**PROGRAMA QUE SE CONSEGUE USAR AS VARIÁVEIS. NO CASO DA LINGUAGEM C, AS VARIÁVEIS SÃO DIVIDIDAS QUANTO AO ESCOPO EM TRÊS TIPOS. ASSINALE A OPÇÃO QUE APRESENTA ESSES TRÊS TIPOS:**

**A)** Variáveis por valor, variáveis por referência e variáveis por argumento

**B)** Variáveis locais, variáveis globais e parâmetros formais

**C)** Parâmetros formais, parâmetros informais e sem parâmetro

**D)** Parâmetros formais, variáveis internas e variáveis externas

**E)** Parâmetros atuais, parâmetros reais e parâmetro formais

**2. ANALISE O SEGUINTE CÓDIGO IMPLEMENTADO NA LINGUAGEM C.**

```
INT SOMA(INT A, INT B) {
INT S;
S = A + B;
RETURN S;
}
INT MAIN() {
INT X=5, Y=10;
INT Z ;
Z = SOMA(X, Y);
PRINTF("%D", Z);
RETURN(0);
}
```

**ASSINALE A OPÇÃO CORRETA:**

**A)** As variáveis a e b são variáveis globais.

**B)** As variáveis x e y são parâmetros formais.

**C)** A variável z está vinculada à função é main e por isso é uma variável global.

**D)** O escopo de vinculação da variável s está definido pela função soma.

**E)** Esse programa não tem variável local, todas são globais.

---

## GABARITO

**1. Nas linguagens de programação, as variáveis são vinculadas aos seus valores em tempo de execução. E o escopo de uma vinculação é o conjunto de trechos de um programa que se consegue usar as variáveis. No caso da linguagem C, as variáveis são divididas quanto ao escopo em três tipos. Assinale a opção que apresenta esses três tipos:**

A alternativa **"B "** está correta.

No caso da linguagem C, as variáveis são divididas quanto ao escopo em três tipos: Variáveis locais, variáveis globais e parâmetros formais.

**2. Analise o seguinte código implementado na linguagem C.**

```
int soma(int a, int b) {
int s;
s = a + b;
return s;
}
int main() {
int x=5, y=10;
int z ;
z = soma(x, y);
printf("%d", z);
return(0);
}
```

**Assinale a opção correta:**

A alternativa **"D "** está correta.

A variável s está declarada dentro da função soma, portanto é uma variável local da função soma e seu escopo está vinculado à execução da função.

# CONCLUSÃO

# CONSIDERAÇÕES FINAIS

Iniciamos este tema conceituando modularização, abordando a evolução da programação de código para uma programação estruturada e elencando as vantagens − como a reutilização de código − ao se utilizar esse modelo.

Como vimos, a modularização divide o código do programa em sub-rotinas que podem ser procedimentos ou funções. Definimos e apontamos, ainda, a principal diferença entre elas.

Além disso, entendemos o funcionamento da passagem por parâmetro tanto por valor como por referência e identificamos o escopo das variáveis quando as distinguimos entre locais e globais.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.

# REFERÊNCIAS

SCHILDT, H. **C, completo e total**. 3. ed. São Paulo: Makron Books, 1996, cap. 5, p. 113.

---

# EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, assista ao canal:

**Linguagem C Programação Descomplicada.**

---

# CONTEUDISTA

Daisy Albuquerque

## 🔗 CURRÍCULO LATTES